WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

A FAST AND SIMPLE ALGORITHM FOR THE MAXIMUM FLOW PROBLEM

R. K. Ahuja
and
J. B. Orlin

Sloan W.P. No. 1905-87 June 1987

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

A FAST AND SIMPLE ALGORITHM FOR THE MAXIMUM FLOW PROBLEM

R. K. Ahuja
and
J. B. Orlin

Sloan W.P. No. 1905-87 June 1987

A Fast and Simple Algorithm for the Maximum Flow Problem

Ravindra K. Ahuja* and James B. Orlin
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA. 02139 , USA

Abstract

We present a simple $O(nm + n^2 \log U)$ sequential algorithm for the maximum flow problem with n nodes, m arcs, and a capacity bound of U among arcs directed from the source node. Under the practical assumption that U is polynomially bounded in n , our algorithm runs in time $O(nm + n^2 \log n)$. This result improves the current best bound of $O(nm \log (n^2/m))$, obtained by Goldberg and Tarjan , by a factor of log n for non-sparse and non-dei.se networks without using any complex data structures.

Subject Classification
484. A Fast and Simple Algorithm for the Maximum Flow Problem

* On leave from Indian Institute of Technology, Kanpur - 208 016 , INDIA

The maximum flow problem is one of the most fundamental problems in network flow theory and has been investigated extensively. This problem was first formulated by Ford and Fulkerson [1956] who also solved it using their well-known augmenting path algorithm. Since then a number of algorithms have been developed for this problem, as tabulated below. Here n is the number of nodes, m is the number of arcs, and U is an upper bound on the integral arc capacities.

#	Due to	Running Time
1	Ford and Fulkerson [1956]	$O(nm U)$
2	Edmonds and Karp [1972]	$O(nm^2)$
3	Dinic [1970]	$O(n^2m)$
4	Karzanov [1974]	$O(n^3)$
5	Cherkasky [1977]	$O(n^2m^{1/2})$
6	Malhotra, Kumar and Maheshwari [1978]	$O(n^3)$
7	Galil [1978]	$O(n^{5/3}m^{2/3})$
8	Galil and Naamad [1978]	$O(nm \log^2 n)$
9	Shiloach and Vishkin [1982]	$O(n^3)$
10	Sleator and Tarjan [1983]	$O(nm \log n)$
11	Tarjan [1984]	$O(n^3)$
12	Gabow [1985]	$O(nm \log U)$
13	Goldberg [1985]	$O(n^3)$
14	Goldberg and Tarjan [1986]	$O(nm \log (n^2/m))$

Edmonds and Karp [1972] showed that the Ford and Fulkerson [1956] algorithm runs in time $O(nm^2)$ if flows are augmented along shortest paths from source to sink. Independently, Dinic [1970] introduced the concept of shortest path networks, called *layered* networks, and obtained an $O(n^2m)$ algorithm. This bound was improved to $O(n^3)$ by Karzanov [1974] who introduced the concept of *preflows* in a layered network. A *preflow* is similar to a flow except that the amount flowing into a node may exceed the amount flowing out of a node. Since then, researchers have improved the complexity of Dinic's algorithm for sparse networks by devising sophisticated data structures. Among these contributions, Sleator and Tarjan's [1983] dynamic tree is the most attractive from a worst case point of view.

The algorithms of Goldberg [1985] and of Goldberg and Tarjan [1986] were a novel departure from these approaches in the sense that they did not construct layered networks. It contains the essence of Karzanov's preflow method but does not maintain layered networks. Their algorithm maintains a preflow and proceeds by pushing flows to nodes estimated to be closer to the sink. To estimate which nodes are closer to the sink, it maintains a distance label for each node that is a lower bound on the length of a shortest augmenting path to the sink. Distance labels are a better computational device than layered networks since these labels are simpler to understand, easier to manipulate, and easier to use in a parallel algorithm. Moreover, by cleverly implementing the dynamic tree data structure, Goldberg and Tarjan obtained the best computational complexity for sparse as well as dense networks. Bertsekas [1986] recently developed an algorithm for the minimum cost flow problem that simultaneously generalizes both the Goldberg–Tarjan algorithm and Bertsekas [1979] auction algorithm for the assignment problem.

For problems with arc capacities polynomially bounded in n , our maximum flow algorithm is an improvement of Goldberg and Tarjan's algorithm and uses concepts of scaling introduced by Edmonds and Karp [1972] for the minimum cost flow problem and later extended by Gabow [1985] for other network optimization problems. The bottleneck operation in the straightforward implementation of Goldberg and Tarjan's algorithm is the number of "non-saturating pushes" which is $O(n^3)$. However, they reduce the computational time to $O(nm \log (n^2/m))$ by a very clever application of the dynamic tree data structure. We show that the number of non-saturating pushes can be reduced to $O(n^2 \log U)$ by using "excess scaling." Our algorithm performs $\log U$ scaling iterations; each scaling iteration requires $O(n^2)$ non-saturating pushes if we push flows from nodes with *sufficiently large* excesses to nodes with *sufficiently small* excesses while never allowing the excesses to become *too large.* Consequently, the computational time of our algorithm is $O(nm + n^2 \log U)$. Under the reasonable assumption that $U = O(n^{0(1)})$ (i.e., it is polynomial in n), the algorithm runs in time $O(nm + n^2 \log n)$ which improves the bound of Goldberg and Tarjan's algorithm by a factor of $\log n$ for both non-sparse and non-dense networks, i.e., networks for which $m = O(n^{1+\varepsilon})$ and $m = \Omega(n^{1+\varepsilon})$ for some ε with $0 < \varepsilon < 1$. Moreover, our algorithm is easy to implement (and is much more efficient in practice), since it requires only elementary data structures with little computational overhead.

Our algorithm is computationally attractive even if U is not $O(n^{0(1)})$ and the arc capacities are exponentially large numbers. In this case, the uniform model of computation, in which all arithmetic operations take $O(1)$ steps, is arguably inappropriate. It is more realistic to adopt the logarithmic model of computation (as described by Cook and Reckhow [1973]) which counts the number of *bit* operations. In this model, most arithmetic operations take $O(\log U)$ steps than $O(1)$ steps.

Using the logarithmic model of computation and modifying our algorithm slightly to speed up arithmetic operations on large integers, we obtain an $O(nm \log n + n^2 \log n \log U)$ algorithm for the maximum flow problem. The corresponding time bound for the Goldberg-Tarjan algorithm is $O(nm \log (n^2/m) \log U)$. Hence as U becomes exponentially large, our algorithm surpa Goldberg and Tarjan's algorithm by a factor of m/n in the non-dense c n other words, the relative worst case performance of our algorithm i singly superior as U increases in size. Our results on this logarithmic model of computation will be presented in Ahuja and Orlin [1987a].

Our maximum flow algorithm is difficult to make "massively parallel" since we push flow from one node at a time. Nevertheless, with $d = \lceil m/n \rceil$ parallel processors we can obtain an $O(n^2 \log Ud)$ time bound. Under the assumption that $U = O(n^{0(1)})$, the algorithm runs in $O(n^2 \log n)$ time, which is comparable to the best available time bounds obtained by Shiloach and Vishkin [1982] and Goldberg and Tarjan [1986] using *n parallel processors*. Thus, our algorithm has an advantage in situations for which parallel processors are at a premium. Our work on the parallel algorithms will be presented in Ahuja and Orlin [1987b].

1. Notation

Let $G = (N, A)$ be a directed network with a positive integer capacity u_{ij} for every arc $(i, j) \in A$. Let $n = |N|$ and $m = |A|$. The source *s* and sink *t* are two distinguished nodes of the network. We assume without loss of generality that the network does not contain multiple arcs. We further assume that for every arc $(i, j) \in A$, an arc (j, i) is also contained in A, possibly with zero capacity. Let $U = \max_{(s,j) \in A} \{u_{sj}\}$.

A *flow* is a function $x\colon A \to R$ satisfying

$$\sum_{\{j:\ (j,i)\in A\}} x_{ji} - \sum_{\{j:\ (i,j)\in A\}} x_{ij} = 0 \ , \text{ for all } i \in N - \{s,t\}, \tag{1}$$

$$\sum_{\{j:\ (j,t)\in A\}} x_{jt} = v \ , \tag{2}$$

$$0 \le x_{ij} \le u_{ij} \ , \quad \text{for all } (i,j) \in A \ , \tag{3}$$

for some $v \ge 0$. The maximum flow problem is to determine a flow x for which v is maximized.

A *preflow* x is a function $x\colon A \to R$ which satisfies (2) , (3), and the following relaxation of (1):

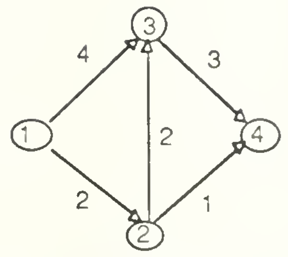
$$\sum_{\{j:\ (j,i)\in A\}} x_{ji} - \sum_{\{j:\ (i,j)\in A\}} x_{ij} \ge 0 \ , \text{ for all } i \in N - \{s,t\}. \tag{4}$$

The algorithms described in this paper maintain a preflow at each intermediate stage.

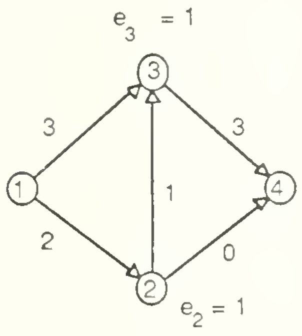
For a given preflow x , we define for each node $i \in N - \{s,t\}$, the *excess*

$$e_i = \sum_{\{j:\ (j,i)\in A\}} x_{ji} - \sum_{\{j:\ (i,j)\in A\}} x_{ij} \ .$$

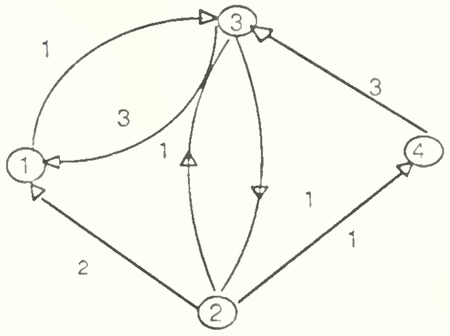
A node with positive excess is referred to as an *active node*. The *residual capacity* of any arc $(i, j) \in A$ with respect to a given preflow x is given by $r_{ij} =$ [illegible] $- x_{ij} + x_{ji}$. The network consisting only of arcs with positive residua [illegible] cities is referred to as the *residual network.* Figure 1 illustrates thes [illegible] tions.



a. Network with arc capacities. Node 1 is the source. Node 4 is the sink. (Arcs with zero capacities are not shown)



b. Network with a preflow x



c. The residual network with residual arc capacities

Figure 1. Illustrations of preflow and residual network.

A *distance function* $d: N \rightarrow Z^+$ for a preflow x is a function from the set of nodes to the non-negative integers . *We say that the distance function is valid if it also satisfies the following two conditions:*

C1. d(t

C2. d(i,) + 1 , for every arc $(i, j) \in A$ with $r_{ij} > 0$.

Our algorithm maintains a valid distance function at each iteration, and labels each node i with the corresponding value d(i). The distance d(i) is a lower-bound on the length of any path from node i to t in the residual network. This fact is easy to demonstrate using induction in the value d(i). If for each i, the distance label d(i) equals the minimum length of any path from i to t in the residual network, then we call the distance label *exact*. For example, in Figure 1(c), d = (0 , 0 , 0 , 0) is a valid distance label, though d = (3, 1, 2, 0) represents the exact distance labels.

We define the *arc adjacency list* A(i) of a node $i \in N$ as the set of arcs directed out of the node i , i.e., $A(i) := \{(i, k) \in A : k \in N\}$. Note that our adjacency list is a set of arcs rather than the more conventional definition of a list as a set of nodes.

All logarithms in this paper are assumed to be of base 2 unless stated otherwise.

2. The Preflow-Push Algorithms

The *preflow-push algorithms* for the maximum flow problem maintain a preflow at every step and proceed by pushing the excess of nodes closer to the sink. The first preflow-push algorithm is due to Karzanov [1974]. Tarjan [1984] has suggested a simplified version of this algorithm. The recent algorithms of Goldberg [1985] and Goldberg and Tarjan [1986] are based on ideas similar to those presented in Tarjan [1984] but use distance labels to direct flows closer to the sink instead of constructing layered network. We thus call their algorithm the *distance-directed* preflow-push algorithm. In this section, we review the basic features of this algorithm, which for the sake of brevity, we shall simply refer to as the preflow-push algorithm. Here we describe the 1-phase version of the preflow-push algorithm presented by Goldberg [1987]. The results in this section are due to Goldberg and Tarjan [1986].

All operations of the preflow-push algorithm are performed using only local information. At each iteration of the algorithm (except at the initialization and at the termination) the network contains at least one active node, i.e., a (non-source) node with positive excess. The goal of each iterative step is to choose some active node and to send its excess "closer" to the sink, with closer being judged with respect to the current distance labels. If excess flow at this node cannot be sent to nodes with smaller distance labels, then the method increases the distance label of the node. The algorithm terminates when the network contains no active nodes. The preflow-push algorithm uses the following subroutines:

PRE-PROCESS. On each arc $(s, j) \in A(s)$, send u_{sj} units of flow. Let $d(s) = n$ and $d(t) = 0$. Let $d(i) = 1$ for each $i \neq s$ or t. (Alternatively, the distance label for each node $i \neq s, t$ can be determine a breadth first search on the residual network starting at node t.)

SELECT. Select an active node $i \neq s, t$ (i.e., $e_i > 0$).

PUSH(i). Select an arc $(i, j) \in A(i)$ with $r_{ij} > 0$ and $d(i) = d(j) + 1$. Send $\delta = \min \{e_i, r_{ij}\}$ units of flow from node i to j.

We say that a push of flow on arc (i, j) is *saturating* if $\delta = r_{ij}$, and *non-saturating* otherwise.

RELABEL(i). Replace d(i) by $\min\{ d(j) + 1 : (i, j) \in A(i) \text{ and } r_{ij} > 0 \}$.

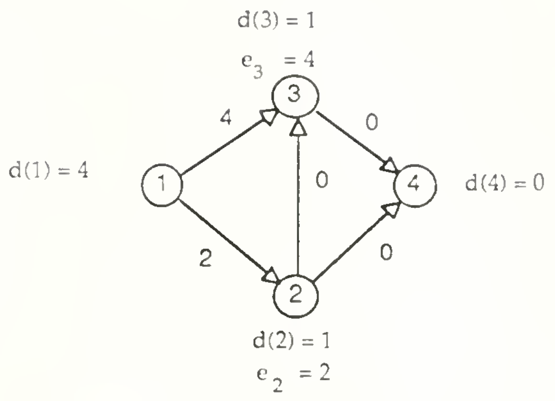
Figure 2 contains the generic version of the preflow-push algorithm.

```
begin
        PREPROCESS
        while  there is an active node in  $N - \{s, t\}$  do
        begin
                SELECT {let i denote the node selected};
                If  there is an arc  $(i, j) \in A(i)$  with  $r_{ij} > 0$  and
                        $d(i) = d(j) + 1$  then  PUSH(i)
                else RELABEL(i)
        end;
end;
```

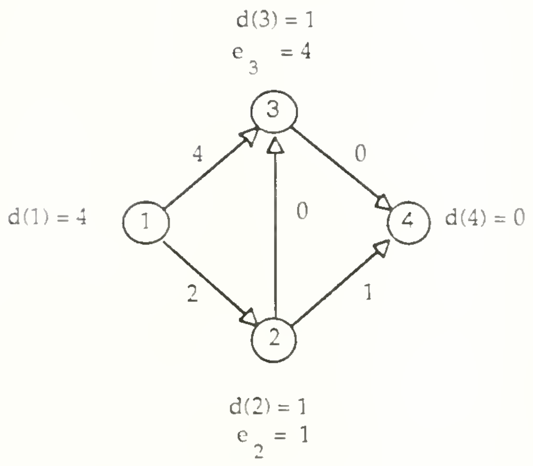
Figure 2. The preflow-push algorithm

Figure 3 illustrates the steps PUSH(i) and RELABEL(i) as applied to the network in Figure 1(a). Figure 3(a) specifies the preflow determined by PRE–PROCESS. The SELECT step selects node 2 for examination. Since arc (2, 4) has residual capacity $r_{24} = 1$ and $d(2) = d(4) + 1$, the algorithm performs a (saturating) push of value $\delta = \min\{2, 1\}$ units. The push reduces the excess of node 2 to 1. Arc (2, 4) is deleted from the residual network and arc (4, 2) is added to the residual network. Since node 2 is still an active node, it can be selected again for further pushes. The arcs (2, 3) and (2, 1) have positive residual capacities, but they do not satisfy the distance condition. Hence the algorithm performs RELABEL(2) ,and gives node 2 a new distance $d'(2) = \min \{d(3) + 1 , d(1) + 1\} = \min \{2, 5\} = 2$.

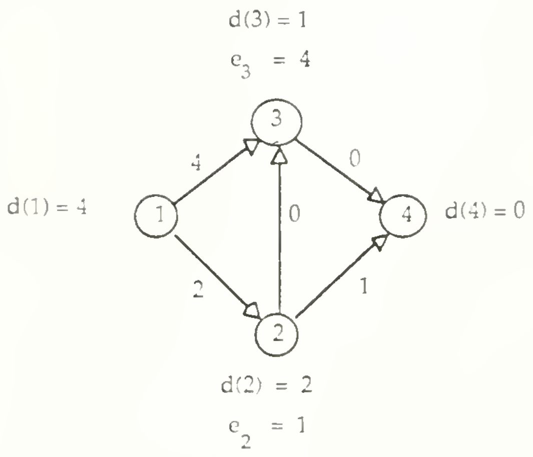
The pre-process step accomplishes several important tasks. First, it causes the nodes adjacent to s to have positive excess, so that we can select some node with positive excess. (Otherwise, the algorithm could not get started.) Second, by saturating arcs incident to s , the feasibility of setting $d(s) = n$ is immediate. Third, since $d(s) = n$ is a lower bound on the length of the minimum path from s to t , there is no path from s to t . Since distances in d are non–decreasing (see Lemma 1 to follow), we are also guaranteed that in subsequent iterations the residual network will never contain a directed path from s to t, and so there can never be any need to push flow from s again.



(a) Tr ork with a preflow after the pre-processing step.



(b) After the execution of step PUSH(2).



(c) After the execution of step RELABEL(2).

Figure 3. Illustrations of Push and Relabel steps.

In our improvement of the preflow-push algorithm, we need a few of the results given in Goldberg and Tarjan [1986]. We include some of their proofs in order to make this presentation more self-contained.

Lemma 1. The generic preflow-push algorithm maintains valid distance labels at each step. Moreover, at each relabel step the distance label of a node strictly increases.

Proof. First note that the pre-process step constructs valid distance labels. Assume inductively that the distance function is valid prior to an operation, i.e., satisfies the validity conditions C1 and C2. A push operation on the arc (i, j) may create an additional arc (j, i) with $r_{ji} > 0$, and an additional condition $d(j) \le d(i) + 1$ needs to be satisfied. This validity conditions remain satisfied since $d(i) = d(j) + 1$ by the property of the push operation. A push operation on arc (i, j) might delete this arc from the residual network, but this does not affect the validity of the distance function. During a relabel step, the new distance label of node i is $d'(i) = \min\{d(j) + 1 : (i, j) \in A(i) \text{ and } r_{ij} > 0\}$, which is consistent with the validity conditions. The relabel step is performed when there is no arc $(i, j) \in A(i)$ with $d(i) = d(j) + 1$ and $r_{ij} > 0$. Hence,
$d(i) < \min\{d(j) + 1 : (i, j) \in A(i) \text{ and } r_{ij} > 0\} = d'(i)$, thereby proving the second part of the lemma. ■

Lemma 2. At any stage of the preflow-push algorithm, each node with positive excess is connected to node s by a directed path in the residual network.

Proof. By the flow decomposition theory of Ford and Fulkerson [1962], any preflow x can be decomposed with respect to the original network G into non-negative flows along (i) paths from source s to t, (ii) paths from s to active nodes pl and (iii) the flows around directed cycles. Let i be an active node in th flow x of G. Thus there must be a path P from s to i in the flow dec tion of x. Then the reversal of P (P with the orientation of each arc reversed) is in the residual network G', and hence there is a path from i to s in G'. ■

Corollary 1. For each node $i \in N$, $d(i) < 2n$.

Proof. The last time that node i was relabeled, it had a positive excess, and hence the residual network contained a path of length at most $n-1$ from i to s. The fact that $d_s = n$ and condition C2 imply that $d_i \le d_s + n - 1 < 2n$. ■

Corollary 2. The number of relabel steps is less than $2n^2$.

Proof. Each relabel step increases the distance label of a node by at least one, and by Corollary 1 no node can be relabeled more than at most $2n$ times. ■

Corollary 3. The number of saturating pushes is no more than nm.

Proof. Suppose that arc (i, j) becomes saturated at some iteration (at which $d(i) = d(j) + 1$). Then no more flow can be sent on (i, j) until flow is sent back from j to i (at which $d'(j) = d'(i) + 1 \ge d(i) + 1 = d(j) + 2$); This flow change cannot occur until $d(j)$ increases by at least 2. Thus by Corollary 1 arc (i, j) can become saturated at most n times, and the total number of arc saturations is no more than nm. (Recall that we assume that (i, j) and (j, i) are both in A, so the number of arcs in the residual network is no more than m.) ■

Lemma 3. The number of non-saturating pushes is at most $2n^2m$.

Proof. See Goldberg and Tarjan [1986] .

Lemma 4. The algorithm terminates with a maximum flow.

Proof. When the algorithm terminates, each node in $N - \{s, t\}$ has zero excess; so the final preflow is a feasible flow. Further, since the distance labels satisfy the conditions C1 and C2 and $d(s) = n$, it follows upon termination the residual network contains no directed path from s to t . This condition is the classical termination criteria for the maximum flow algorithm of Ford and Fulkerson [1962] . ■

The potential bottleneck operation in the preflow based algorithms due to Karzanov [1974] , Tarjan [1984], and Goldberg and Tarjan [1986] is the number of non-saturating pushes, which dominates the number of saturating pushes for the following reason: The saturating pushes cause structural changes -- they delete saturated arcs from the residual network. This observation leads to a bound of $O(nm)$ on the number of saturating pushes --no matter in which order they are performed. The non-saturating pushes do not change the structure of the residual network and seem more difficult to bound. Goldberg [1985] reduced the non-saturating pushes to $O(n^3)$ by examining the nodes in a specific order, and Goldberg and Tarjan [1986] reduced the average time per non-saturating push by using a sophisticated data structure. In the next section we show that by using scaling, we can dramatically reduce the number of non-saturating pushes to $O(n^2 \log U)$.

3. The Scaling Algorithm

Our maximum flow algorithm improves the generic preflow-push algorithm of Section 2, and uses "excess scaling" to reduce the number of non-saturating pushes from $O(n^2m)$ to $O(n^2 \log U)$. The basic idea is to push flow from active nodes with *sufficiently large* excesses to nodes with *sufficiently small* excesses while never letting the excesses become *too large.*

The algorithm performs $K = \lceil \log U \rceil + 1$ scaling iterations. For a scaling iteration, the *excess-dominator* is defined to be the least integer Δ that is a power of 2 and satisfies $e_i \le \Delta$ for all $i \in N$. Further , a new scaling iteration is considered to have begun whenever Δ decreases by a factor of 2. In a scaling iteration we assure that each non-saturating push sends at least $\Delta/2$ units of flow and the excess-dominator does not increase. To ensure that each non-saturating push has value at least $\Delta/2$, we consider only nodes with excess more than $\Delta/2$; and among these nodes with large excess, we select a node with minimum distance label .

We show that after at most $4n^2$ non-saturating pushes, the excess–dominator decreases by a factor of at least 2 and a new scaling iteration begins. After at most K scaling iterations, all node excesses drop to zero and we obtain a maximum flow.

In order to select an active node with excess more than $\Delta/2$ and with a minimum distance label among such nodes, we maintain the lists $\text{LIST}(r) := \{i \in N : e_i > \Delta/2 \text{ and } d(i) = r\}$ for each $r = 1, \dots, n-1$. These lists can be maintained in the form of either a linked stack or a linked queue (see, for example, Aho, Hopcroft and Ullman [1974]), which enables insertion and deletion of elements in O(1) time.) The variable *level* indicates the smallest index r for which LIST(r) is non-empty.

As per Goldberg and Tarjan, we use the following data structure to select efficiently the eligible arc for pushing flow out of a node. We maintain with each node i a list, A(i), of arcs directed out of it. Arcs in each list can be arranged arbitrarily, but the order once decided remains unchanged throughout the algorithm. Each node i has a *current arc* (i, j) which is the current candidate for pushing flow out of i. Initially, the current arc of node i is the first arc in its arc list. This list is examined sequentially and whenever the current arc is found to be ineligible for pushing flow, the next arc in the arc list is made the current arc. When the arc list is completely examined, we say that the next arc is null. At this point, the node is relabeled and the current arc is again the first arc in its arc list.

The algorithm can be described formally as follows:

```
procedure   MAX-FLOW;
begin
        PRE-PROCESS;
        K: =   $\log U\rceil$ ;
        for  k: = 0 to K  do
                begin
                        $\Delta = 2^{K-k}$
                        for each  $i \in N$   do   if  $e_i > \Delta/2$  then  add  i  to  LIST(d(i));
                        level : = 1 ;
                        while  level < 2n  do
                            if  LIST(level) = ø   then  level: = level + 1
                            else
                                  begin
                                       select a node  i  from  LIST(level);
                                       PUSH/RELABEL(i)
                                  end;
                end;
end;
```

```
procedure PUSH/RELABEL(i);
begin
    found: = false ;
    let (i, j) be the current arc of node i;
    while found = false  and  (i, j) $\neq$ null  do
            if d(i) = d(j) + 1  and  $r_{ij} > 0$  then found: = true
            else  replace the current arc of node i by the next arc (i, j);

    if  found = true  then
            begin  {push}
                push  min{$e_i$ , $r_{ij}$ , $\Delta - e_j$}  units of flow on arc (i, j) ;
                update residual capacities and the excesses;
                if (the updated excess)  $e_i \leq \Delta/2$ , then delete node i
                        from  LIST(d(i));
                if  j $\neq$ s  or  t  and (the updated excess) $e_j > \Delta/2$ , then
                        add node j  to  LIST(d(j))  and set level: =  level – 1;
            end
    else
            begin {relabel}
                delete node i  from LIST(d(i));
                update  d(i): = min{d(j) + 1 ;  (i, j) $\in$ A(i) and  $r_{ij} > 0$} ;
                add node i  to  LIST(d(i))  and let the current arc of
                the node be the first arc of  A(i);
            end;
end;
```

4. Complexity of the Algorithm

In this section , we show that the distance–directed preflow-push algorithm with scaling correctly computes a maximum flow in $O(nm + n^2 \log U$ [illegible] me.

Lemma 5. The [illegible] thm satisfies the following two conditions:

C3. Each non-saturating push from a node i to a node j sends at least $\Delta/2$ units of flow.

C4. No excess increases above Δ . (i.e., the excess-dominator does not increase subsequent to a push.)

Proof. For every push on arc (i, j) we have $e_i > \Delta/2$ and $e_j \le \Delta/2$, since node i is a node with smallest distance label among nodes whose excess is more than $\Delta/2$, and $d(j) = d(i) - 1 < d(i)$ by the property of the push operation. Hence , by sending $\min \{e_i , r_{ij} , \Delta - e_j \} \ge \min \{\Delta/2 , r_{ij}\}$ units of flow, we ensure that in a non-saturating push the algorithm sends at least $\Delta/2$ units of flow. Further, the push operation increases e_j only . Let e'_j be the excess at node j subsequent to the push. Then $e'_j = e_j + \min \{ \Delta/2 , r_{ij} , \Delta - e_j \} \le e_j + \Delta - e_j \le \Delta$. All node excesses thus remain less than or equal to Δ .

While there are other ways of ensuring that the algorithm always satisfies the properties stated in the conditions C3 and C4, pushing flow from a node with excess greater than $\Delta/2$ and with minimum distance among such nodes is a simple and efficient way of enforcing these conditions.

With properties C3 and C4, the push operation may be viewed as a kind of "restrained greedy approach." Property C3 ensures that the push from i to j is sufficiently large to be effective. Property C4 ensures that the maximum infeasibility of flow conservation never exceeds Δ during an iteration. In particular, rather than greedily getting rid of all its excess, node i shows some restraint so as to prevent e_j from exceeding Δ. (It is possible that the maximum excess increases during a push. In particular, if the maximum excess is between Δ and $\Delta/2$, it may increase to Δ.)

Keeping the maximum excess as low as in C4 may be very useful in practice as well as in theory. Its major impact is to "encourage" flow excesses to be distributed fairly equally in the network. This distribution of flows should make it easier for nodes to send flow towards the sink. (Alternatively, if a number of nodes were to send flow to a single node j , it is likely that node j would not be able to send the accumulated flow directly to the sink; much of the excess of node j would have to be returned from where it came.)

Lemma 6. If each push satisfies conditions C3 and C4, then the number of non-saturating pushes per scaling iteration is at most $8n^2$.

Proof. Consider the potential function $F = \sum_{i \in N} e_i \, d(i)$.

The initial value of F is bounded by $2n^2\Delta$ because e_i is bounded by Δ and d(i) is bounded by 2n . When the algorithm examines node i , one of the following two cases must apply:

Case 1. The algorithm is unable to find an arc along which flow can be pushed. In this case no arc (i, j) satisfies $d(i) = d(j) + 1$ and $r_{ij} > 0$ and the distance label of de i goes up by $\delta \geq 1$ units. The increase in F is thus bounded by δ. Since the initial value of d(i) and all its increases are bounded by 2n, the increases in F due to relabelings of nodes are bounded by $2n^2 \Delta$.

Case 2. The algorithm is able to identify an arc on which flow can be pushed and so it performs either a saturating or a non-saturating push. In either case, F decreases. Moreover, each non-saturating push sends at least $\Delta/2$ units of flow to a node with smaller distance label with a corresponding decrease in F of at least $\Delta/2$ units. Since the initial value of F plus the increases in F cannot exceed $4n^2 \Delta$, this case cannot occur more than $8n^2$ times. (A more careful analysis leads to a bound of $4n^2$.) ■

Theorem 1. The procedure Max-Flow performs $O(n^2 \log U)$ non-saturating pushes.

Proof. The initial value of the excess-dominator Δ is $2^{\lceil \log U \rceil} \leq 2U$. By Lemma 5 , the value of the excess-dominator decreases by a factor of 2 within $8n^2$ non-saturating pushes and a new scaling iteration begins. After $1 + \lceil \log U \rceil$ such scaling iterations, $\Delta < 1$ and by the integrality of the flows $e_i = 0$ for all $i \in N - \{s, t\}$. The algorithm thus obtains a feasible flow, which by Lemma 4 must be a maximum flow.

Theorem 2. The complexity of the maximum flow algorithm is $O(nm + n^2 \log U)$.

Proof: The complexity of the algorithm depends upon the number of executions of the **while** loop in the main program. In each such execution either a PUSH/RELABEL(i) step is performed or the value of the variable **level** increases. Each execution of the procedure PUSH/RELABEL(i) results in one of the following outcomes:

Case 1. A push is performed. Since the number of saturating pushes are $O(nm)$ and the non-saturating pushes are $O(n^2 \log U)$, this case occurs $O(nm + n^2 \log U)$ times.

Case 2. The distance label of node i goes up. By Corollary 2, this outcome can occur $O(n)$ times for each node i. Computing the new distance label requires examining arcs in $A(i)$. Since each arc in A is examined $O(n)$ times, the total effort for all of the $O(n^2)$ distance increases is $O(nm)$.

Thus the algorithm calls the procedure PUSH/RELABEL(i) $O(nm + n^2 \log U)$ times. The effort needed to find an arc to perform the push operation is $O(1)$ plus the number of times the current arc of node i is replaced by the next arc in A(i). After $|A(i)|$ such replacements for node i, Case 2 occurs and distance label of node i goes up. Thus, the total effort needed is $\sum_{i \in N} 2n\,|A(i)| = O(nm)$ times plus the number of PUSH/RELABEL(i) operations. This is clearly

$O(nm + n^2 \log U)$. The lists LIST(i) are stored as linked stacks and queues, hence addition and deletion of any element takes $O(1)$ time. Consequently, updating these lists is not a bottleneck operation. Finally, we need to bound the number of increases of the variable *level*. In each scaling iteration, *level* is bounded above by $2n - 1$ and bounded below by 1. Hence its number of increases per scaling iteration is bounded by the number of decreases plus $2n$. Further, *level* can decrease only when a push is performed and in such a case it decreases by 1. Hence its increases over all scaling iterations are bounded by the number of pushes plus $(2n \log U)$, which is again $O(nm + n^2 \log U)$. ■

5. Refinements

As a practical matter, several modifications of the algorithm might improve its actual execution time without affecting its worst case complexity. We characterize the modifications as one of the following three types:

1. Modify the scale factor.
2. Allow some non-saturating pushes of small amount.
3. Try to locate nodes disconnected from the sink.

The algorithm in the present form uses a scaling factor of 2, i.e., it reduces the excess-dominator by a factor 2 in the consecutive scaling iterations. However, in practice it might be better to scale the excess-dominator by some other fixed integer factor $\beta \geq 2$. The excess-dominator will be the least power of β that is no less than the excess flow at any node, and property C3 becomes

C3'. Each non-saturating push from a node i to a node j sends at least Δ/β units of flow.

The procedure maxflow can easily be altered to incorporate the β scale factor by letting $LIST(j) = \{i: d(i) > \Delta/\beta\}$. The algorithm can be shown to run in $O(nm + \beta n^2 \log_\beta U)$ time. Clearly, from the worst case point of view any fixed value of β is optimum. The best choice of the value of β in practice should be determined empirically.

Our algorithm as stated selects a node with $e_i > \Delta/2$ and performs a saturating or a non-saturating push. We could, however, keep pushing the flow out of this node until either we perform a non-saturating push of value $\Delta/2$ or reduce its excess to zero. This variation might produce many saturating pushes from the node and even decrease its excess below $\Delta/2$.

Also, the algorithm as stated sends at least $\Delta/2$ units of flow during every non-saturating push. The same complexity of the algorithm is obtained if for some fixed $r \geq 1$, one out of every $r \geq 1$ non-saturating pushes sends at least $\Delta/2$ units of flow.

One potential bottleneck in practice is the number of relabels. In particular, the algorithm "recognizes" that the residual network contains no path from node i to node t only when d(i) exceeds $n - 2$. Goldberg [1987] suggested that it may be desirable occasionally to perform a breadth first search so as to make the distance labels exact. He discovered that a judicious use of breadth first search could dramatically speed up the algorithm.

An alternative approach is to keep track of the number n_k of nodes whose distance is k. If n_k decreases to 0 after any relabel, then each node with distance greater than k is disconnected from the sink in the residual network. (Once node j is disconnected from the sinks, it stays disconnected since the shortest path from j to t is nondecreasing in length.) We would avoid selecting such nodes until all nodes become disconnected from the sink. At this time, the excesses of active nodes are sent back to the source.

6. Conclusions and Summary

Our improvement of the distance-based preflow-push algorithm has several advantages over other algorithms for the maximum flow problem.

First, our algorithm is the most efficient algorithm for the maximum flow problem under the reasonable assumption that U is polynomially bounded in n . (We can relax this assumption on U if we modify the algorithm slightly and adopt the more realistic logarithmic model of computation as discussed in [1987a].)

Second, our model has several intuitively appealing features that are not guaranteed in other preflow-push methods. We always try to push flow from a node with a large excess. (Gabow's scaling version of Dinic's algorithm always selects an augmentation along a path with a large capacity.) This feature accounts for much of the efficiency of his method. Also, we never allow too much excess to accumulate at a node. In addition, our algorithm relies on very simple data structures with little computational overhead. This feature contrasts dramatically with the algorithms that rely on dynamic trees.

Third, our algorithm is a novel approach to combinatorial scaling algorithms. In the previous scaling algorithms developed by Edmonds and Karp [1972] , Rock [1980] , and Gabow [1985] , scaling involved a sequential approximation of either the cost coefficients or the capacities and right-hand-sides. (e.g., we would first solve the problem with the costs approximated by $c/2^T$ for some integer T. We would then reoptimize so as to solve the problem with c approximated by $c/2^{T-1}$. We would then reoptimize for the problem with c approximated by $c/2^{T-2}$, and so forth.) Our scaling method does not fit into this standard framework. In Ahuja and Orlin[1987a], we describe an alternate framework for scaling algorithms which includes all of the previous scaling algorithms as well as the algorithm described here and the algorithms of Goldberg and Tarjan [1987] for the minimum cost flow problem, and the algorithms of Gabow and Tarjan [1987] for the assignment and related problems.

Fourth, our algorithm appears to be quite efficient in practice. In Ahuja and Orlin [1987c], we empirically test a number of different algorithms for the maximum flow problem. The algorithm presented here appears from preliminary results to be comparable to the best other maximum flow algorithm for a range of distributions. In addition, our algorithm involves only simple data structures so that it is relatively easy to encode.

Fifth, our algorithm can be implemented efficiently on a parallel computer with a small number of parallel processors. In particular, suppose that $d = m/n$, and that $\log U = O(\log n)$. Then our algorithm runs in $O(n^2 \log n)$ time on a parallel random access machine (PRAM) using only d parallel processors. This time bound is comparable to the time bound achieved by Goldberg and Tarjan [1986] using n parallel processors. Thus we introduce a n/d improvement in utilization of computational resources. In addition, our model of computation is less restrictive. These results are discussed in Ahuja and Orlin [1987b].

Acknowledgements

We wish to thank John Bartholdi and Tom Magnanti for their suggestions which led to improvements in the presentation. This research was supported in part by the Presidential Young Investigator Grant 8451517-ECS of the National Science Foundation.

REFERENCES

Aho, A.V. , J.E. Hopcroft and J.D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA.

Ahuja, R.K., and J.B. Orlin. 1987a. Scaling Algorithms for Combinatorial Optimization Problems and the Logarithmic Model of Computation. Work in progress.

Ahuja, R.K., and J.B. Orlin. 1987b. A Fast Parallel Maximum Flow Algorithm Using Few Parallel Processors. Work in progress.

Ahuja, R.K., and J.B. Orlin. 1987c. Numerical Investigation of Maximum Flow Algorithms. Work in progress.

Bertsekas, D.P. 1979. A Distributed Algorithm for the Assignment Problem. Unpublished Manuscript.

Bertsekas, D.P. 1986 . Distributed Relaxation Methods for Linear Network Flow Problems. *Proceedings of the 25th IEEE Conference on Decision and Control* , Greece .

Cherkasky, R.V. 1977 . Algorithm of Construction of Maximal Flow in Networks with Complexity of $O(V^2 \sqrt{E})$ Operation, *Mathematical Methods of Solution of Economical Problems 7,* 112-125 (in Russian).

Cook, S.A., and R.A. Reckhow. 1973. Time Bounded Random Access Machines . *J. of Comput. System Sci.* 7 , 354 - 375 .

Dinic, E.A. 1970. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation, *Soviet Math. Dokl. 11* , 1277-1280.

Edmonds, J., and R.M. Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. Assoc. Comput Mach. 19,* 248-264.

Ford, L.R., and D.R. Fulkerson. 1956. Maximal Flow through a Network. *Can. J. Math. 8,* 399-404.

Ford, L.R., and D.R. Fulkerson. 1962. *Flows in Networks.* Princeton University Press, Princeton, New Jersey.

Gabow, H.N. 1985. Scaling Algorithms for Network Problems. *J. of Comput. System Sci. 31*, 148-168.

Gabow, H.N., and R.E. Tarjan. 1987. Faster Scaling Algorithms for Graph Matching. Research Report, Computer Science Dept., Princeton University, Princeton, New Jersey.

Galil, Z. 1980. An O($V^{5/3}$ $E^{2/3}$) Algorithm for the Maximal Flow Problem, *Acta Informatica 14* , 221-242.

Galil, Z. and A. Naamad. 1980. An O($EV^{5/3}$ $\log_2$ V) Algorithm for the Maximum Flow Problem. *J. Comput. System Sci.* 21 , 203-217.

Goldberg, A.V. 1985 . A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291 , Laboratory for Computer Science, M.I.T. , Cambridge, MA.

Goldberg, A.V., and R.E. Tarjan. 1986. A New Approach to the Maximum Flow Problem. *Proceedings of the Eight Annual ACM Symposium on the Theory of Computing.*

Goldberg, A.V., and R.E. Tarjan. 1987. Solving Minimum Cost Flow Problem by Successive Approximation. *Proceedings of the Ninth Annual ACM Symposium on the Theory of Computing.*

Goldberg, A.V. 1987. Efficient Graph Algorithms for Sequential and Parallel Computers. Unpublished Ph.D. Dissertation, Laboratory for Computer Science, M.I.T., Cambridge, MA.

Karzanov, A.V. 1974. Determining the Maximal Flow in a Network by the Method of Preflows, *Soviet Math. Dokl. 15* , 434-437.

Malhotra, V.M., M. Pramodh Kumar, and S.N. Maheshwari. 1978. An O($|V|^3$) Algorithm for Finding Maximum Flows in Networks. *Inform. Process. Lett. 7* , 277-278.

Rock, H. 1980. Scaling Techniques for Minimal Cost Network Flows, *Discrete Structures and Algorithms,* Eds. V. Page, Carl Hanser, München, 181-191.

Shiloach, Y. 1978. An O(n I $\log^2$ I) Maximum-Flow Algorithm. Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford University, Stanford, CA.

Shiloach, Y. , and U. Vishkin. 1982. An O(n^2 log n) Parallel Max-Flow Algorithm. *J. Algorithms 3* , 128-146.

Sleator, D.D. 1980. An O(nm log n) Algorithm for Maximum Network Flow, Tech. Rep. STAN-CS-80-831, Computer Science Dept., Stanford University, Stanford, CA.

Sleator, D.D., and R.E. Tarjan. 1983. A Data Structure for Dynamic Trees, *J. Comput. System Sci. 24*, 362-391.

Tarjan, R.E. 1984. A Simple Version of Karzanov's Blocking Flow Algorithm, *Operations Research Lett. 2* , 265-268.

4534 053